Center for Applied Autonomous Sensor Systems (AASS)
Örebro University, School of Science and Technology
SE-70182 Örebro
Sweden

OOP Lab II: OOAD

Welcome to OOP Lab II! This lab contains several exercizes of different complexity and on different topics.

Note: this lab should be done in groups, so look at the blackboard to see what group you are in. Ask the instructor if you have not been assigned to any group!

Part 0: Connecting to git-e server from ThinLinc

Some of us periodically have problems connecting to the git server (gir-e.oru.se) with SSH key. That is normally caused by excessive uncuccessfull login attempts so the server blocks SSH access from offending IP addresses. If you are blocked, try connecting to the repository using HTTPS protocol.

To avoid blocking:

- Thinlink is whitelisted in the git server settings, so you should not have any blocking problemd from Thinlink, unless you are using wrong key or passphrase.
- Make sure you can also connect to git-e server using your standard login/password over HTTPS.
- Make sure to not make too many login attempts!

Part 1: Methods and Tools.

In this lab, we will be mostly drawing diagrams and writing cases. There are several possible options to do this:

- Paper and Pencil. Back in the days, they were considered highly reliable and usable.
- Whiteboard. Ask instructor for markers if you have non yourself.
- Draw.IO (https://www.draw.io/) allows to draw object, interaction, and class diagrams using standard building blocks.
- PlantUML – is a language and a plugin for NetBeans (go to NetBeans – Tools – Plugins to install it) which allows to build UML diagrams using mark-up language.

PlantUML is a recommended tool in this lab.

To install PlantUML into your NetBeans, go to NetBeans plugin portal and download PlantUML plugin (http://plugins.netbeans.org/plugin/49069/plantuml). Then, in NetBeans, open plugins andf install it. Restart NetBeans.

Part 1: OOAD Process for a larger app.

The process of OOAD, which we looked at on the previous lectures, allows to create a basic structure of the future software, already filled with classes, their attributes and behaviors. In fact, some UML drawing programs allow to generate the source code of the class structure automatically for a number of OOP languages. Some programs even allow to reconstruct the UML diagram from the source code. The very important note to remember is that it is the **class diagram**, which **defines the future software**, but **not vice versa**. Starting coding too early might give some quick results in the beginning of a project, but is likely to cause many design problems later, when the time is much more critical.

Speaking about problems in software, there are different kinds of problems, which lead to different outcomes. Syntactic errors are the easiest to spot, because the software most likely won't compile and run. Algorithmic errors usually result in wrong program behaviors and can be spotted by using unit and functional testing. Code smells are not necessary problems, but can lead to problems in future. Fortunately, code smells have some typical "signatures" and can be spotted by experienced developers. Finally, **design errors** are hardest to spot, because they do not result in any compilation errors or runtime exceptions, are very hard to spot just by looking at the source code. But design errors become a significant obstacle in the future development, because they can make software rigid and fragile.

Fortunately, over the long history of software engineering, many developers had to solve similar problems and came up with similar solutions. These are usually called Principles of Object-Oriented Design.

The goal of today's lab is to develop a class diagram and a sequence diagram of an Automated Teller Machine (ATM)[1]. An ATM has a display, a keyboard, a card reader, and a cash tray. It is able to provide a user with the following services: balance check, transfer, withdrawal, and deposit.

Please, follows the four step OOAD process: find object, find attribute, find behaviours, group. As usual, start with developing use cases (it's a good idea to make an individual use case for every operation).

Hint: In the use case, consider the bank API as a separate actor from which you can request some data regarding the cardholder.

Hint: Try to keep as lees information as possible regarding the cardholder for security reasons.

Relationships

Remember to be clear about relationships between objects (and classes):

1 Since we rapidly moving to the cashless society, you might not know what ATMs were. For additional info please refer to this Wikipedia page: https://en.wikipedia.org/wiki/Automated_teller_machine

- Association (knows): object A can call a member of object B, but does not have it as its own member or has no other relationship with it.
- Aggregation (has): object A includes object B as its member, but B is created outside.
- Composition (has): objects A includes object B and limits its lifetime (B is created and removed by A).
- Generalization (is): object B is derived from object A (e.g. Car can be derived from Vehicle or Player can be derived from Human).
- Realization (is): object B is derived from object A and A is an interface (has no internal implementation by itself).

Note: not all software packages conform to the UML notation standards.

Design Evaluation

First of all, remember the rule: **Design for Change**!

When you think that your design has already something meaningfull in it, start applying design principles to see if your design violates any of them. The design principles in this course are as follows:

- Single responsibility principle: a class should have only a single responsibility, or a class should have only a signle reason to change.
- Open/closed principle: "software entities … should be open for extension, but closed for modification."
- Liskov substitution principle: "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." Or, if class B is a subclass (inherited) from A, then any object of class A should be replaceable by object of class B.
- Interface segregation principle: "many client-specific interfaces are better than one general-purpose interface." Or, the entioty should not be forced to know more than needed for its functioning.
- Dependency inversion principle: one should "depend upon abstractions, [not] concretions." Or, high level modules should not depend on low level modules. Instead, both should conform to abstraction (e.g. interface).

Assessment

You are free to ask for advise during the lab. You can also ask for an offline feedback on your design. For that, submit your design to the teacher and TA in blackboard upon completing the task.